Introduction to Computation and Problem Solving

Class 28:
Binary Search Trees

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Binary Search Trees

- **In the previous lecture, we defined the concept of binary search tree as a *binary tree* of nodes containing an *ordered key* with the following additional property. The left subtree of every node (if it exists) must only contain nodes with keys less than or equal to the parent and the right subtree (if it exists) must only contain nodes with keys greater than or equal to the parent.**
- **We saw that performing an *inorder* traversal of such a tree is would visit each node in order.**

2

Goals

- **We are going to implement a binary search tree with the usual data structure operations and then critique the implementation.**
- **Generalizations of the binary search tree drive many important applications, including commercial databases.**
- **The implementation will give you a feel for tree methods that tend to be more "visual" or "topological" than array or list implementations.**

3

Keys and Values

- **If binary search trees are ordered, then they must be ordered on some *key* possessed by every tree node.**
- **A node might contain nothing but the *key*, but it's often useful to allow each node to contain a *key* and a *value*.**
- **The *key* is used to look up the node. The *value* is extra data contained in the node indexed by the *key*.**

4

Maps

- **Such data structures with key/value pairs are usually called *maps*.**

- **As an example, consider the entries in a phone book as they might be entered in a binary search tree. The subscriber name, last name first, serves as the *key*, and the phone number serves as the *value*.**

5

Key and Value Type

- **Sometimes you may not need the value. But if one node has a value, you probably want to supply one for every node.**

- **We also haven't specified the type of the key and value. They can be of any *class* you want, but all nodes should possess keys that are instances of the same class and values that are instances of the same class.**

- **Keys and values do *not* have to be the same type. Keys and values will be handled as `Objects` by our tree implementation except when we need to compare them.**

6

Duplicate Keys

- **Do we allow the tree to contain nodes with identical keys?**
- **There is nothing in the concept of a binary search tree that prevents duplicate keys.**
- **In the interface that we will implement, we use the keys to access the values. If we allowed duplicate keys, we wouldn't be able to distinguish between multiple nodes possessing the same key.**
- **Thus, in this implementation, duplicate keys will not be allowed. Duplicate values associated with different keys are legal.**

7

SortedMap Interface

```
public interface SortedMap {
  // keys may not be null but values may be
  public boolean isEmpty();
  public void clear();
  public int size();
  public Object get( Object key );
  public Object firstKey();
  public Object lastKey();
  public Object remove( Object key );
  public Object put( Object key, Object value );
  public MapIterator iterator();
}
```

8

SortedMap Notes

- `get(), firstKey(), lastKey()` **all return the appropriate values and leave the key value pair in the map.**
- `firstKey()` **and** `lastKey()` **return** `null` **if the map is empty**
- `remove()` **returns the value if the key is present; otherwise it returns** `null`
- `put():` **if the key is not in the map, insert the new key/value pair; if the key is already present, return the old value, and replace the old value with the new one in the map**

9

MapIterator Interface

```
public interface MapIterator
{
    public boolean hasNext();
    public Object next()
        throws NoSuchElementException;
    public Object remove()
        throws IllegalStateException;
    public Object key()
        throws IllegalStateException;
    public Object getValue()
        throws IllegalStateException;
    public Object setValue( Object value )
        throws IllegalStateException;
}
```

10

Comparator

If keys are arbitrary objects, how can you order them?

The user can supply his own Comparator

```
public interface Comparator
{
    // Neither o1 nor o2 may be null
    public int compare(Object o1, Object o2);
}
// e.g., compare( Integer(5), Integer(7)) < 0
//       compare( Integer(5), Integer(5)) == 0
```

11

StringComparator

```
public class StringComparator
       implements Comparator
{
  public int compare(Object o1, Object o2)
  {
    return ((String) o1).compareTo((String) o2);
  }
}

BinarySearchTree btree =
  new BinarySearchTree( new StringComparator() );
```

12

Comparable Interface

- **In recent versions of the JDK, the wrapper classes for builtin types all implement a separate `Comparable` interface:**

```
public interface Comparable
{
    public int compareTo(Object o);
}
```

- **Since `String` implements the `Comparable` interface, we don't need to implement the special `StringComparator`.**
- **The `compareTo()` method returns `int`s with the same sense as the `Comparator` interface.**

13

Implementing BinarySearchTree

- **We will have separate tree and node classes for `BinarySearchTree`.**
- **We will use a private static inner class for the node class, `Branch`, just as we did for the `SLink` class in `SLinkedList`.**

```
public class BinarySearchTree
  implements SortedMap
{
    private Branch root = null;
    private int length = 0;
    private Comparator comparator = null;
```

14

Branch Inner Class

- **`Branch` holds links to its parent node as well as left and right subtrees.**
- **This will make some operations like deleting a node much more efficient.**

```
static private class Branch
{
   private Branch left = null;
   private Branch right = null;
   private Branch parent = null;
   private Object key;
   private Object value = null;
```

15

Simple Methods

```
public BinarySearchTree( Comparator c )
{ comparator = c; }

public BinarySearchTree()
{ this( null ); }

public void clear() {
   root = null;
   length = 0;
}

public boolean isEmpty() {
   return ( root == null );
}
```

16

8

firstKey() Method

```
public Object firstKey() {
   if ( root == null ) return null;
   else return root.first().key;
}
In Branch:
Branch first() {
   Branch f = this;
   while ( f.left != null )
   { f = f.left; }
   return f;
}
```
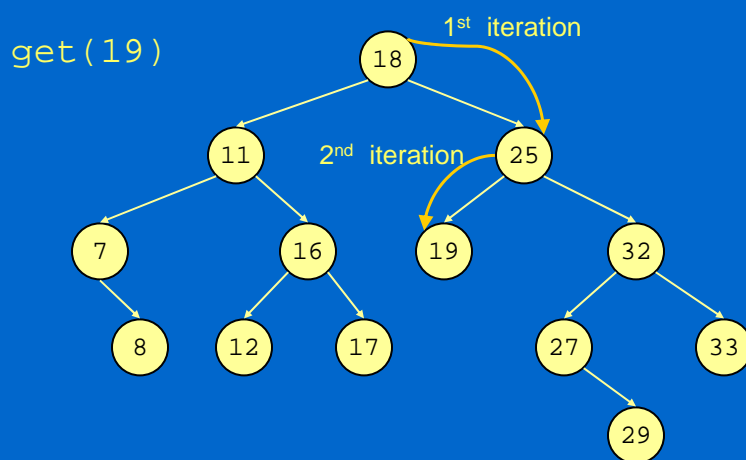
17

firstKey() in Action
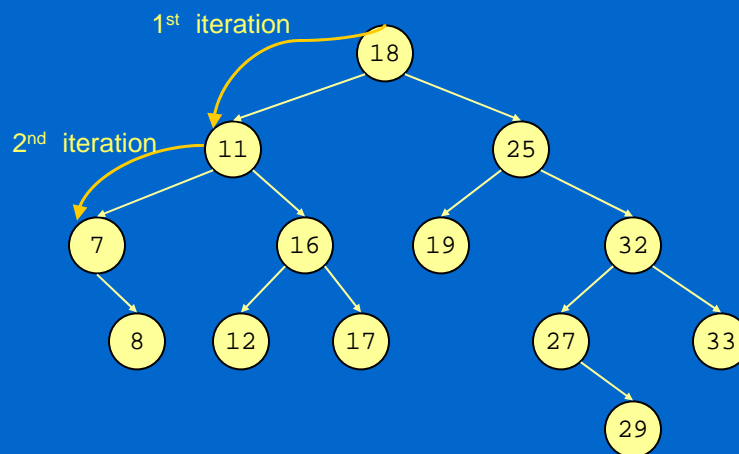
18

get() Method

```
public Object get( Object k ) {
    Branch b = find( k );
    return ( b == null ) ? null : b.value;
}

private Branch find( Object k ) {
    Branch b = root;
    int c;
    while (b != null && (c=compare( k, b.key )) != 0) {
        if ( c < 0 ) b = b.left;
        else b = b.right;
    }
    return b;
}
```

19

get() in Action

get(19)

1st iteration

2nd iteration

20

10

compare() Method

```
private int compare(Object k1, Object k2) {
   if ( k1 == null || k2 == null )
      throw new IllegalArgumentException(
        "Null key now allowed" );
   // if there is a comparator, use it
   if ( comparator != null )
      return comparator.compare( k1, k2 );
   else  {
      return ((Comparable)k1).compareTo(k2);
   }
}
```
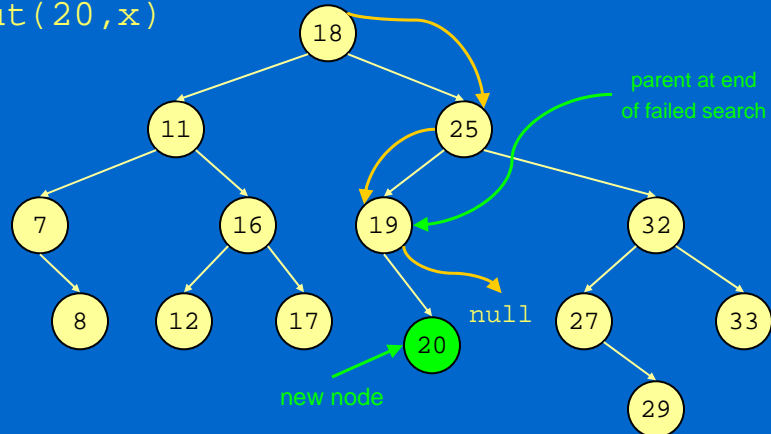
21

put() Strategy

- **Our strategy is to look for the key supplied as an argument.**
- **If we find it, we return the old value after replacing it with the new one.**
- **While searching for the node, shadow our search by keeping a reference to the parent of the current node.**
- **If we don't find the node, the search ends when we arrive at a null branch.**
- **In this case, the parent reference will identify the parent of the new inserted node.**

22

put() in Action

put(20,x)

```
                    18
           11                25
      7         16      19         32
          8    12  17  20  null  27    33
                     new node        29
```

parent at end
of failed search

null

new node

23

put() Method, 1

```
public Object put(Object k, Object v)
{
   Branch p = null; Branch b = root; int c = 0;
   // find node with key k retaining ref p to parent
   while ( b != null ) {
       p = b; c = compare( k, b.key );
       if ( c == 0 ) {
          // found it; insert new value, return old
          Object oldValue = b.value; b.value = v;
          return oldValue;
       } else if ( c < 0 )
           b = b.left;
       else
           b = b.right;
   }
      24
```

12

put() Method, 2

```
// Key k doesn't exist in tree;
// insert new node below p(arent)
Branch newBranch = new Branch( k, v, p );
length++;
if ( p == null ) {
    root = newBranch;
} else {
// c still holds last comparison
    if ( c < 0 )
      p.left = newBranch;
    else
      p.right = newBranch;
}
return null;
}
    25
```
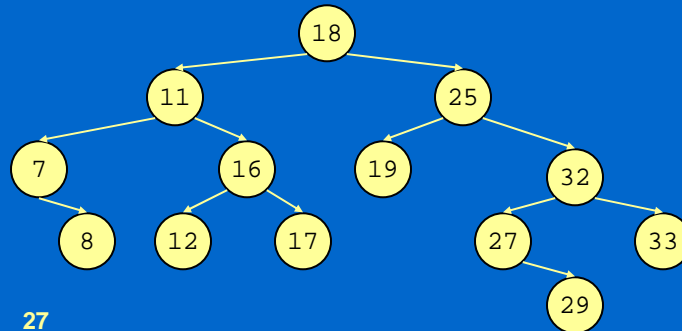
delete() Strategy

- **remove() calls delete() to do most of the work.**
- **When we delete a node from a binary search tree, we must ensure that the resulting structure (1) is still a tree, and (2) the tree still obeys the rule of a binary search tree.**
- **The rule requires that for every node, the keys in the left subtree if present precede the key of the node which must precede the keys in the right subtree.**
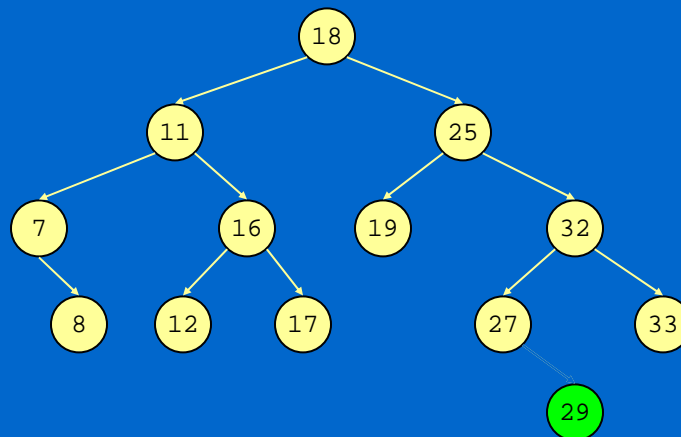
26

13

delete() Cases, 1

There are three deletion cases we must consider:
1. The deleted node has no children, e.g., node 29 below.
2. The deleted node has one child, e.g., node 7 below.
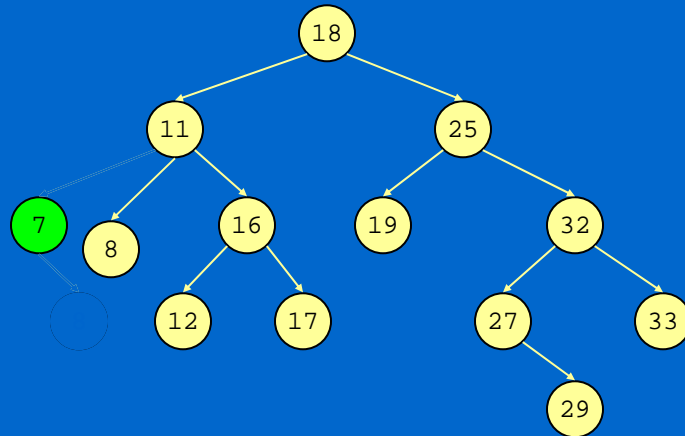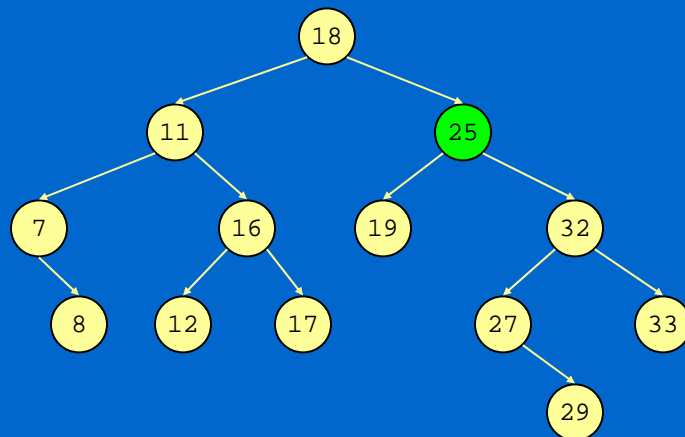3. The deleted node has two children, e.g., node 25 below

27

delete(), No Children

28

14

delete(), 1 Child

18
11   25
7   8   16   19   32
12   17   27   33
29

29

delete(), 2 Children

18
11   25
7   16   19   32
8   12   17   27   33
29

30

15

delete(), 2 Children Strategy

- **In the last case, that of two children, the problem is more serious since the tree is now in three parts.**
- **The solution starts by realizing that we can minimize the ordering problem by stitching the tree back together using the node immediately preceding or following the deleted node in inorder sequence. These are known as the *predecessor* and *successor* nodes.**
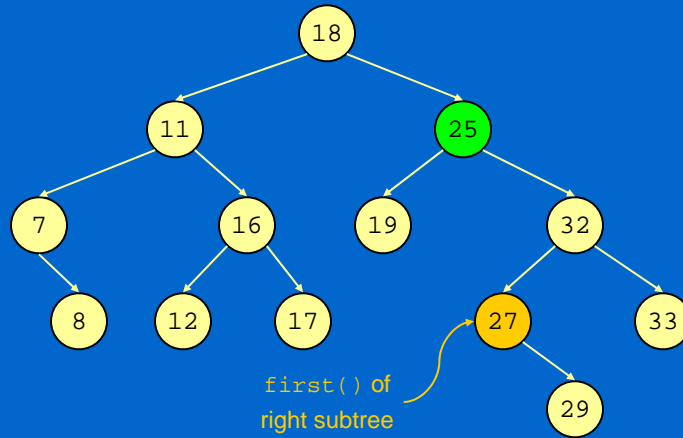
31

Using the Successor Node

- **Let's choose the successor node.**
- **The successor of a node with a right subtree will be the `first()` node of that subtree.**
- **If we replace the deleted node by its successor, then all ordering conditions will be met. But what about the successor's children.**
- **The successor of a node with two children can have at most one child (a right child). If it had a left subtree, the node couldn't be the successor since members of the left subtree follow the deleted node but precede the successor. Since the successor can have at most one subtree, we can move the successor up to replace the node we want to remove and relink its right subtree, if present, as in the second case above.**
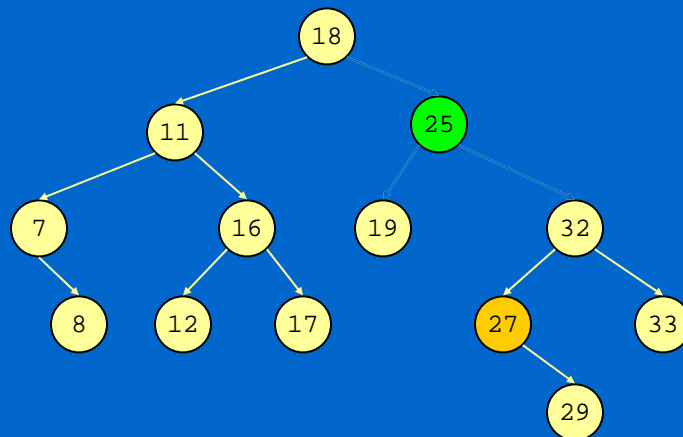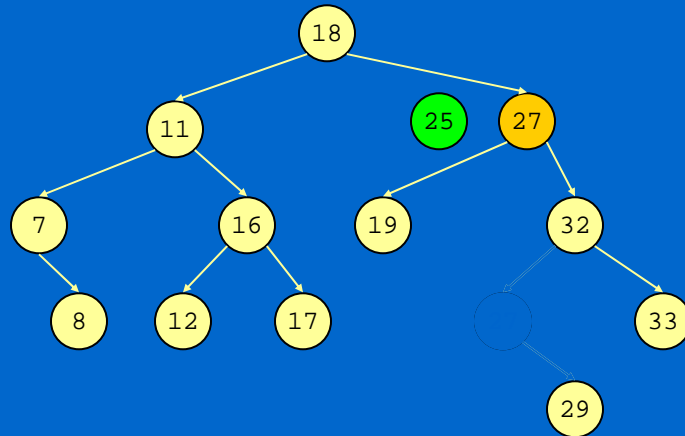
32

delete(), **find successor**

18

11   25

7   16   19   32

8   12   17   27   33

first() of
right subtree

29

33

delete(), **replace successor 1**

18

11   25

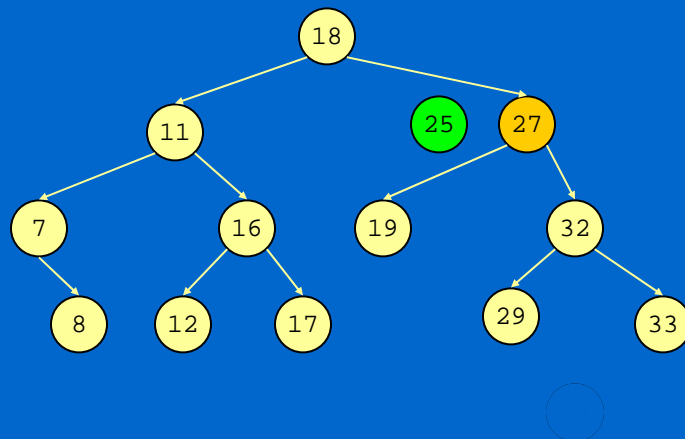7   16   19   32

8   12   17   27   33

29

34

17

delete(), replace successor 2

35

delete(), replace successor 3

36

The Efficiency of Binary Search

- It makes intuitive sense that the basic operations on a binary search tree should require *O(h)* time where h is the height of the tree.
- It turns out that the height of a *balanced* binary tree is roughly $\log_2(n)$ where n is the number of elements if the tree remains approximately balanced.
- It can be proved that if keys are randomly inserted in a binary search tree, this condition will be met, and the tree will remain adequately enough balanced so that search and insertion time will approximate *O(log n)*.
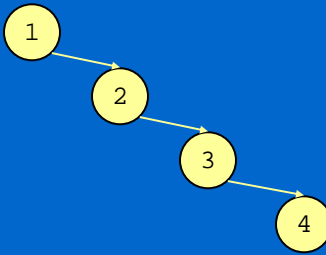
37

Tree Balance

- There are some extremely simple and common cases, however, where keys will not be inserted in random order.
- Consider what will happen if you insert keys into a search tree from a sorted list. The tree will assume a degenerate form equivalent to the source list and search and insertion times will degrade to *O(n)*.
- There are many variants of trees, e.g., red-black trees, AVL trees, B-trees, that try to solve this problem by rebalancing the tree after operations that unbalance it.

38